Certifying the Absence of Buffer Overflows

Sagar Chaki
Scott Hissam

September 2006

Predictable Assembly from Certifiable Components Initiative

Technical Note
CMU/SEI-2006-TN-030

The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

Acknowledgment

We are grateful to James Ivers, Kurt Wallnau, and the other members of the Predictable Assembly from Certifiable Components (PACC) Initiative at the Carnegie Mellon® Software Engineering Institute SEI for supporting us at various stages of this research. We thank John Wilander and Tim Leek for allowing us access to their benchmark suites. We also thank Kendra Kratciewicz and Robert Seacord for making their benchmarks and examples publicly available. Finally, we are grateful to Noam Zeilberger for providing critical feedback on an early draft of this report.

[0] ® Carnegie Mellon is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

Abstract

Despite increased awareness and efforts to reduce buffer overflows, they continue to be the cause of most software vulnerabilities. In large part, these problems are due to the widespread use of unsafe library routines among programmers. For reasons like efficiency, such routines will continue to be used, even during the development of mission-critical and safety-critical software systems. Effective certification techniques are needed to ascertain whether unsafe routines are used in a safe manner.

This report presents a technique for certifying the safety of buffer manipulations in C programs. The approach is based on two key ideas: (1) using a certifying model checker to automatically verify that a buffer manipulation is safe and (2) validating the resulting invariant and proving it with a decision procedure based on Boolean satisfiability. This report also discusses the advantages and limitations of the approach with respect to today's existing solutions for buffer-overflow detection. Experimental results are presented that position the technique favorably against other static overflow-detection tools and indicate that the procedure can complement and augment these tools from a purely verification perspective.

1 Introduction

A recent study funded by the U.S. Department of Commerce's National Institute of Standards and Technology (NIST) concluded that software bugs, or errors, are so prevalent and so detrimental that they cost the U.S. economy an estimated $60 billion annually [NIST 02]. A substantial portion of programmatic errors ultimately manifest themselves as software vulnerabilities. For example, it is estimated that "hacker attacks cost the world economy a whopping $1.6 trillion in 2000" and "U.S. virus and worm attacks cost $10.7 billion in the first three quarters of 2001 [Jarzombek 04]." This problem is further highlighted by the increasing number of attacks that exploit such vulnerabilities. For example, The CMU CERT® Coordination Center[1] reported 76,404 attack incidents in the first half of 2003, approaching the total of 82,094 for all of 2002 in which the incident count was nearly four times the 2000 total.[2] In fact, CERT statistics often understate the problem by counting all related attacks as a single incident.

Buffer overflows are widely recognized [Cowan 00] to be the prime source of vulnerabilities in commodity software. For example, the CodeRed [3] worm that caused $2.1 billion in global damage in 2001 exploited a buffer overflow in Windows [Jarzombek 04]. In addition, Wagner and colleagues report, on the basis of CERT advisories, that "buffer overruns account for up to 50% of today's vulnerabilities, and this ratio seems to be increasing over time" [Wagner 00]. Buffer overflows are problematic because attackers use them to execute arbitrary code (such as a shell) with administrative privileges. For example, a common strategy is to overwrite a program's activation record in order to redirect its control flow to any desired point. As such, buffer overflows are extremely dangerous and can lead to catastrophic system compromises and failures.

Broadly speaking, a buffer overflow occurs when a piece of data D is written to a buffer B such that the size of D is greater than the allocated size of B. In the case of a type-safe language or a language with explicit bounds checking (such as Java), an overflow leads to either a (static) type error or a (runtime) exception. Unfortunately, the vast majority of commercial and legacy software is written in unsafe languages (such as C or C++) that allow buffers to be overflowed with impunity. Due to efficiency and other reasons, the unsafe use of these languages is unlikely to abate. In fact, the overflow problem persists even when only "safer" library routines, such as `fgets`, `snprintf`, and `strncpy`, are used because programmers pass incorrect array bounds information to them. Therefore, it is important that we develop techniques to guard against buffer overflows, while still allowing low-level buffer accesses.

In this report, we present an *automated* approach that uses formal proofs as a means of assurance against the possibility of buffer overflows. Given a C program P and a target buffer operation O, our technique[4] leads to one of the following outcomes:

[1] CERT and CERT Coordination Center are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

[2] http://www.sei.cmu.edu/pacc/files/DOD-whitepaper.pdf.

[3] http://www.cert.org/advisories/CA-2001-19.html.

[4] Our approach allows for multiple target operations. However, we use a single target in this report for simplicity.

- It produces a program trace *CE* that potentially leads to a buffer overflow at O. The counterexample *CE* acts as a diagnostic feedback and aids in debugging P.

- It yields a certificate *Cert* that attests to the fact that no buffer overflows can occur while executing O. It also generates annotations for P that can be used to validate *Cert* **without having to trust** either *Cert* or the annotations.

Specifically, our technique achieves its goal in the following step-wise manner:

1. **interpretation.** First, the program P is transformed to a new program $\widehat{P}$ by adding extra code that keeps track of various buffer manipulations. We also add an assertion A that fails if, and only if, a buffer can overflow at O.

2. **verification.** Next, a certifying software model checker, based on automated iterative refinement, verifies that the assertion A cannot fail. If the verification fails, we obtain a counterexample $\widehat{CE}$ that leads to an assertion failure at A. Then, our procedure terminates with a counterexample *CE* that shows how a buffer overflow can occur at O. The counterexample *CE* is constructed by reverse interpreting $\widehat{CE}$. If, on the other hand, the verification succeeds, the certifying model checker also generates an invariant *Inv*, and we proceed to Step 3.

3. **certification.** The invariant *Inv* associates with each control flow point l of $\widehat{P}$ a condition on the variables in scope at l that is true whenever the execution of $\widehat{P}$ reaches l. In this final stage, our procedure verifies that: (a) *Inv* is indeed a valid invariant and (b) *Inv* implies the impossibility of the failure of the assertion A. This verification is achieved by constructing a verification condition VC—a logical formula equivalent to (a) and (b) above—and then proving the validity of VC. To prove VC, we use a decision procedure based on Boolean satisfiability (SAT). Doing so enables us to generate extremely compact proofs. The proof of VC acts as the certificate *Cert*, while *Inv* is used to generate the annotations for P, and our procedure terminates with these artifacts. Note that *Cert* can be validated subsequently by: (a) reconstructing VC using *Inv* and (b) using a proof checker to ensure that *Cert* is indeed a valid proof of VC. More importantly, this procedure does not require us to trust either *Inv* or *Cert*.

We believe that ours is the first approach to use automated iterative refinement and certifying software model checking for buffer-overflow detection. In addition, our technique is the first with the ability not only to find buffer overflows but also to certify their absence. Furthermore, a relatively small trusted computing base (TCB)—comprised of a VC generator and a proof checker—can validate the certificates our approach generates. State-of-the-art software analysis tools are, themselves, complex software artifacts and thus should not be trusted any more than the systems they analyze. Therefore, the elimination of the verification engine from the TCB enhances considerably the confidence we have in our certification process.

We have implemented our technique in the ComFoRT reasoning framework [Chaki 05] and experimented with four suites of publicly available benchmarks. Our experimental results indicate that, in addition to being unique on account of its certification capabilities, our technique is positioned favorably against other static overflow-detection tools and can complement and augment these tools even if we are primarily interested in verification. Further details of our experiments are provided in Section 6. The remaining sections of this report are organized as follows. In Section 2, we survey related work. The interpretation, verification, and certification steps of our approach are presented in detail in Sections 3, 4, and 5, respectively. Finally, we discuss our experimental results in Section 6 and present our conclusions in Section 7.

2 Related Work

In this section, we discuss existing techniques for overflow detection and software certification. Manual approaches are inherently non-scalable; therefore, we focus on procedures that involve a fair amount of automation. A number of approaches for overflow detection are type-theoretic in nature [Shankar 01]. These approaches require that programs be written in a type-safe language and not be applicable to the vast body of legacy and in-production systems that involve type-unsafe languages such as C or C++. Techniques based on simulation or testing suffer from low coverage and are typically unable to provide any reasonable degrees of assurance about critical software systems. Dynamic or runtime buffer-overflow detection schemes [Ruwase 04, Jones 97, Dahn 03, Dhurjati 06] incur performance penalties that can be unacceptable. Even when performance is not a serious issue, we must be assured of the correctness of a system before it is deployed since any failure in real-life would be catastrophic. Such guarantees can be obtained only via static approaches.

A number of static approaches for buffer-overflow detection have been proposed that rely on static analysis of programs. These approaches are usually based on converting the buffer-overflow problem into a constraint-solving problem (such as integer range checking [Wagner 00] or integer linear programming [Ganapathy 03]) or into a static analysis problem on an integer program [Dor 03]. In principle, static analysis amounts to a form of model checking [Schmidt 98] over the control-flow graph (CFG) of a program. However, a CFG is an extremely imprecise model because it retains control-flow information yet ignores other semantic details completely. In practice, false positives plague static analysis based on the CFG.

In the context of buffer-overflow detection, the abundance of false positives means that every probable overflow flagged by static analysis must be manually inspected to ensure that it corresponds to an actual problem and is not an artifact of the imprecise CFG model with no concrete realization. Our overflow-detection technique is also static but based on a paradigm called iterative refinement. We limit the number of false alarms by eliminating them in an automated manner. You can find concrete details regarding the reduction in false positives that our approach achieves in Section 6.

Our approach also builds on a long line of work on formal certification techniques such as proof-carrying code [Necula 97], certifying model checking [Namjoshi 01, Namjoshi 03], and the combination of the latter with iterative-refinement-based software verification [Henzinger 02]. Indeed, the verification and certification stages of our procedure can be viewed as adaptations of our work on SAT-based software certification [Chaki 06] to the certify the absence of buffer-overflow vulnerabilities.

3 Interpretation

In this section, we present the interpretation of target program P to $\widehat{P}$ in detail. To understand our interpretation, let's first agree on the syntax and semantics of programs. For the sake of simplicity, we demonstrate our technique using a programming language with rudimentary constructs. However, we will show that this language is expressive enough to encode arbitrary C programs. In the following sections, for any function f, we write $\mathit{Dom}(f)$ and $\mathit{Range}(f)$ to denote the domain and range of f, respectively.

3.1 Syntax

Let $\mathbb{Z}$ denote the set of variables and Var be a denumerable set of variables. We distinguish between two types of expressions—lvalues and rvalues. Specifically, an lvalue is an rvalue that is also associated with an address in memory. The set of lvalues LV and the set of rvalues RV are defined in the following mutually recursive BNF form:

$$LV := \mathit{Var} \mid \ * RV$$

$$RV := \mathbb{Z} \mid LV \mid \& LV \mid RV \ \mathit{op} \ RV$$

where op is the usual suite of arithmetic, relational, logical, and bitwise operators in C. We do not include the `[ ]` (array index) operator or structures and unions, since they can be expressed via pointer arithmetic and dereferences. We also do not need typecasting, since all variables are integral. Unary operators are left out for simplicity but can be included easily. In the rest of this report, we use z, v, lv, and rv (and their primed versions) to denote elements of $\mathbb{Z}$, Var, LV, and RV, respectively. There are six types of statements in our language: assignments, branches, allocs, frees, calls, and returns—each is described in detail below. In the rest of this report, we use the terms *location, control location, control point*, and *control flow point* synonymously.

- $\mathit{asgn}(lv, rv, rv')$ sets lv to rv and continues execution from location rv'.
- $\mathit{bran}(rv, rv', rv'')$ evaluates the branch condition rv. If the result is zero, execution continues from location rv''; otherwise it continues from rv'.
- $\mathit{malloc}(lv, rv, rv')$ allocates a fresh memory block of rv bytes and sets lv to the first address of that block. Execution continues from location rv'.
- $\mathit{free}(rv, rv')$ frees the block of memory with starting address rv and continues execution from location rv'.
- $\mathit{call}(p, rv)$ calls procedure p and after executing p, continues from location rv.
- ret returns from the current procedure.

A procedure is a tuple $(\mathit{GVar}, \mathit{LVar}, \mathit{Comm}, \mathit{ILoc})$ where

- $GVar \subseteq Var$ and $LVar \subseteq Var$ are the sets of global and local variables, respectively.
- $Comm : \mathbb{Z} \hookrightarrow Stmt$ is a partial function from locations to statements such that (1) $Dom(Comm)$ is finite and (2) if V is the set of variables appearing in the statements associated with $Comm$, $V \subseteq GVar \cup LVar$ and $retpc \notin V$, where $retpc$ is a special variable that stores return locations from procedures.
- $ILoc \in Dom(Comm)$ is the initial location.

For any procedure $p = (GVar, LVar, Comm, ILoc)$, we write $GVar(p)$ to mean $GVar$ and likewise for $LVar$, $Comm$, and $ILoc$. We use $Proc$ to denote the set of all procedures. Finally, a program is a pair (prs, ipr) where, (i) $prs \subseteq Proc$ is a finite set of procedures such that $\forall p \in prs \cdot \forall p' \in prs \cdot p \neq p' \Rightarrow Dom(Comm(p)) \cap Dom(Comm(p')) = \emptyset$ and (ii) $ipr \in prs$ is the initial procedure.

Encoding C Programs. You can use the following conventions to encode arbitrary C programs in our language. Assignments, control flow, and procedure calls are already included. Dynamic memory allocation and deallocation are handled with *malloc* and *free*. Finally, since our language does not support procedure arguments and return values, we use a set of special variables to pass them back and forth between callers and callees.

Modeling Memory. Normally, memory is modeled as a function from addresses to values. However, for overflow detection we model the memory as a set of blocks, each with its own ID. Formally, a memory configuration (memory, for brevity) is a five-tuple $(val, id, base, alloc, size)$, where (1) $val : \mathbb{Z} \hookrightarrow \mathbb{Z}$ maps an address to its value, (2) $id : \mathbb{Z} \hookrightarrow \mathbb{Z}$ maps an address to the ID of the block to which the address belongs, (3) $base : \mathbb{Z} \hookrightarrow \mathbb{Z}$ maps a block ID to the starting address of that block, (4) $alloc : \mathbb{Z} \hookrightarrow \mathbb{Z}$ maps a block ID to the number of bytes allocated for that block, and (5) $size : \mathbb{Z} \hookrightarrow \mathbb{Z}$ maps a block ID to the number of bytes in that block up to and including the first occurrence of the “null” character. In addition, a well-formed memory must satisfy the following conditions:

- $Dom(id) = Dom(val)$
- $Dom(base) = Dom(alloc) = Dom(size) = Range(id)$
- $Dom(val)$ is finite.

Next, we define a formal semantics of our language. If you are more interested in the actual interpretation procedure, skip to Section 3.3.

3.2 Semantics

The empty memory $\mu_\emptyset$ is the memory satisfying the following condition:

$$Dom(val) = Dom(id) = Dom(base) = Dom(alloc) = Dom(size) = \emptyset$$

Note that $\mu_\emptyset$ is well formed. We write *Mem* to denote the set of all memories. For any memory $\mu = (\mathit{val}, \mathit{id}, \mathit{base}, \mathit{alloc}, \mathit{size})$, we write $\mathit{val}(\mu)$ to mean *val* and likewise for *id*, *base*, *alloc*, and *size*. Also, we write $\mathit{Dom}(\mu)$ to mean $\mathit{Dom}(\mathit{val}(\mu))$.

Function Restriction. For any function $f : X \hookrightarrow Y$ and any $X' \subseteq X$, the function $f\downharpoonleft_{X'}: \mathit{Dom}(f) \cap X' \hookrightarrow Y$ is defined as follows: $\forall x \in \mathit{Dom}(f) \cap X' \cdot f\downharpoonleft_{X'}(x) = f(x)$. In other words, $f\downharpoonleft_{X'}$ is obtained by restricting the function f over the domain $\mathit{Dom}(f) \cap X'$.

Function Extension. For any function $f : X \hookrightarrow Y$ and any $X' \subseteq X$, we write $f + X'$ to denote the set of functions $f' : \mathit{Dom}(f) \cup X' \hookrightarrow Y$ such that $f'\downharpoonleft_{\mathit{Dom}(f)\setminus X'} = f\downharpoonleft_{\mathit{Dom}(f)\setminus X'}$. Thus, $f + X'$ is the set of all functions that are obtained by extending the domain of f to include X'. Any function in $f' \in f + X'$ must agree with f over the set $\mathit{Dom}(f) \setminus X'$ and is defined arbitrarily otherwise. To model allocation and deallocation, we define the *new* and *del* operations.

Memory Allocation ($\mathit{new} \subseteq \mathit{Mem} \times \mathbb{Z} \times \mathit{Mem} \times \mathbb{Z}$). Let μ and μ' be any memories and $z \in \mathbb{Z}$ be any integer. Let i be any integer such that $i \notin \mathit{Range}(\mathit{id}(\mu))$. Then, $\forall a \in \mathbb{Z} \cdot ((\mu, z), (\mu', a)) \in \mathit{new}$ (denoted by $(\mu, z) \xrightarrow{\mathit{new}} (\mu', a)$) if the following conditions hold:

- $\forall a' \cdot a \leq a' < a + z \Rightarrow a' \notin \mathit{Dom}(\mu)$
- $\mathit{val}(\mu') \in \mathit{val}(\mu) + \{a' | a \leq a' < a + z\}$
- $\mathit{id}(\mu') \in \mathit{id}(\mu) + \{a' | a \leq a' < a + z\}$
- $\forall a' \cdot a \leq a' < a + z \Rightarrow \mathit{id}(\mu')(a') = i$
- $\mathit{base}(\mu') \in \mathit{base}(\mu) + \{i\}$ and $\mathit{base}(\mu')(i) = a$
- $\mathit{alloc}(\mu') \in \mathit{alloc}(\mu) + \{i\}$ and $\mathit{alloc}(\mu')(i) = z$
- $\mathit{size}(\mu') \in \mathit{size}(\mu) + \{i\}$

Note that $(\mu, z) \xrightarrow{\mathit{new}} (\mu', a)$ iff μ' is derived from μ by allocating a block of z bytes of memory starting at address a.

Memory Deallocation ($\mathit{del} : \mathit{Mem} \times \mathbb{Z} \to \mathit{Mem}$). Let μ be any memory and a be any address. If $a \notin \mathit{Dom}(\mu)$, $\mathit{del}(\mu, a) = \mu$. Thus, deleting an invalid address has no effect on the memory. Otherwise, let $i = \mathit{id}(\mu)(a)$. If $\mathit{base}(\mu)(i) \neq a$, $\mathit{del}(\mu, a) = \mu$. Therefore, deleting an address that is not the starting address of the corresponding memory block leaves the memory unchanged. Otherwise, let $X = \mathit{Dom}(\mu) \setminus \{a' | a \leq a' < a + \mathit{alloc}(\mu)(i)\}$ and $I = \mathit{Dom}(\mathit{base}(\mu)) \setminus \{i\}$. Then, $\mathit{del}(\mu, a) = \mu'$ such that

1. $\mathit{val}(\mu') = \mathit{val}(\mu)\downharpoonleft_X$
2. $\mathit{id}(\mu') = \mathit{id}(\mu)\downharpoonleft_X$
3. $\mathit{base}(\mu') = \mathit{base}(\mu)\downharpoonleft_I$
4. $\mathit{alloc}(\mu') = \mathit{alloc}(\mu)\downharpoonleft_I$
5. $\mathit{size}(\mu') = \mathit{size}(\mu)\downharpoonleft_I$

In other words, to obtain $del(\mu, a)$, the memory block starting at address a from μ is deallocated.

Store. For any set X, we denote the set of all stacks of type X by $stack(X)$. A store $\sigma : Var \to stack(\mathbb{Z})$ is a map from variables to stacks of addresses. The empty store $\sigma_\emptyset$ maps every variable to the empty stack. We write Sto to mean the set of all stores. For any store σ, variable v and address a we write

- $push(\sigma, v, a)$ to mean the store σ' such that for any variable v', if $v' = v$, $\sigma'(v')$ is obtained by pushing a on top of $\sigma(v')$ and otherwise $\sigma'(v') = \sigma(v')$
- $pop(\sigma, v)$ to mean the store σ' such that for any variable v', if $v' = v$, $\sigma'(v')$ is obtained by popping the top element off $\sigma(v')$ and otherwise $\sigma'(v') = \sigma(v')$

Expression Evaluation. We distinguish between the evaluation of lvalues and rvalues and use different notation to denote the two concepts. In essence, an lvalue evaluates to an address, while an rvalue evaluates to a value. Let $\perp$ be a special element not in $\mathbb{Z}$ that denotes an undefined address or value. For any $X \in stack(\mathbb{Z})$, let $top(X)$ evaluate to $\perp$ if X is empty and to the top element of X otherwise. We implicitly extend the domain of any function ϕ from $\mathbb{Z}$ to $\mathbb{Z} \cup \{\perp\}$ by setting $\phi(\perp) = \perp$ and the domain of the C operators to $\mathbb{Z} \cup \{\perp\}$ by letting the result be $\perp$, if either of the operands is $\perp$.

Let σ be a store and $\mu = (val, id, base, alloc, size)$ be a memory. Then, the functions $\langle\sigma, \mu\rangle : LV \to \mathbb{Z} \cup \{\perp\}$ and $[\sigma, \mu] : RV \to \mathbb{Z} \cup \{\perp\}$ are defined as follows:

- $\langle\sigma, \mu\rangle(v) = top(\sigma(v))$
- $\langle\sigma, \mu\rangle(*rv) = [\sigma, \mu](rv)$
- $[\sigma, \mu](z) = z$
- $[\sigma, \mu](lv) = val(\langle\sigma, \mu\rangle(lv))$
- $[\sigma, \mu](\&lv) = \langle\sigma, \mu\rangle(lv)$
- $[\sigma, \mu](rv \ op \ rv') = [\sigma, \mu](rv) \ op \ [\sigma, \mu](rv')$

Memory Update. For any $\sigma \in Sto$, $\mu \in Mem$, $lv \in LV$, and $rv \in RV$, if $\langle\sigma, \mu\rangle(lv) \neq \perp$ and $\langle\sigma, \mu\rangle(lv) \in Dom(\mu)$ and $[\sigma, \mu](rv) \neq \perp$, we write $\mu[lv = rv]$ to denote the memory μ' such that

1. $val(\mu') \in val(\mu) + \{\langle\sigma, \mu\rangle(lv)\}$
2. $val(\mu')(\langle\sigma, \mu\rangle(lv)) = [\sigma, \mu](rv)$
3. $id(\mu') = id(\mu)$
4. $base(\mu') = base(\mu)$
5. $alloc(\mu') = alloc(\mu)$

6. $\mathit{size}(\mu') = \mathit{size}(\mu)$

Note that μ' is uniquely defined, since $\mathit{val}(\mu')(\langle \sigma, \mu \rangle(lv)) = [\sigma, \mu](rv)$ and μ' must agree with μ at other addresses.

Definition 1 (Transition System). *A Transition System (TS) is a triple* $(S, \mathit{Init}, \delta)$ *where: (1)* S *is a set of states, (2)* $\mathit{Init} \subseteq S$ *is the set of initial states, and (3)* $\delta \subseteq S \times S$ *is the transition relation.*

Semantics. The semantics of a program $P = (\mathit{prs}, \mathit{ipr})$ is given as a TS $[\![P]\!] = (S, \mathit{Init}, \delta)$, defined as follows:

- $S = \mathbb{Z} \times \mathit{Sto} \times \mathit{Mem} \cup \{\mathbf{STOP}\}$: a state is a triple consisting of a location, a store, and a memory. The special state **STOP** denotes the termination of the program.
- Let $\cup_{p \in \mathit{prs}} \mathit{GVar}(p) = \{v_1, \ldots, v_n\}$ be the set of global variables of P. Let $\mu_1, \ldots, \mu_n$ be a sequence of memories and $a_1, \ldots, a_n$ be a sequence of addresses such that
  (1) $(\mu_\emptyset, 1) \xrightarrow{new} (\mu_1, a_1)$
  (2) $\forall i \in \{2, ..., n\} \cdot (\mu_{i-1}, 1) \xrightarrow{new} (\mu_i, a_i)$
  (3) $\forall i \in \{1, ..., n\} \cdot \mathit{val}(\mu_n)(a_i) = 0$

  Also, let $\sigma_1, \ldots, \sigma_n$ be the sequence of stores such that
  (1) $\sigma_1 = \mathit{push}(\sigma_\emptyset, v_1, a_1)$
  (2) $\forall i \in \{2, ..., n\} \cdot \sigma_i = \mathit{push}(\sigma_{i-1}, v_i, a_i)$
  Then, $(\mathit{ILoc}(\mathit{ipr}), \sigma_n, \mu_n) \in \mathit{Init}$. In other words, the initial state is obtained by allocating memory for the global variables and initializing them to zero.
- The transition relation is defined with respect to the program statements and is defined next.

Transitions. Let $s = (l, \sigma, \mu)$ be any state. The successors of s according to the transition relation δ depend on the statement at location l and are defined as follows [we write $s \rightarrow s'$ to mean that $(s, s') \in \delta$]:

- Let $D = \cup_{p \in \mathit{prs}} \mathit{Dom}(\mathit{Comm}(p))$. If $l \notin D$, $s \rightarrow \mathbf{STOP}$, else let cp be the unique element of prs such that $l \in \mathit{Dom}(\mathit{Comm}(\mathit{cp}))$. Let $\mathit{st} = \mathit{Comm}(\mathit{cp})(l)$. Thus, cp is the procedure to which the location l belongs and st is the statement at location l. We now consider subcases based on the type of st. In the following, for any lvalue lv (respectively rvalue rv) we write $\langle lv \rangle$ (respectively $[rv]$) to mean $\langle \sigma, \mu \rangle(lv)$ (respectively $[\sigma, \mu](rv)$).
- $\mathit{st} = \mathit{asgn}(lv, rv, rv')$: If $\langle lv \rangle = \bot$ or $\langle lv \rangle \notin \mathit{Dom}(\mu)$ or $[rv] = \bot$ or $[rv'] = \bot$, $s \rightarrow \mathbf{STOP}$. Otherwise, $s \rightarrow ([rv'], \sigma, \mu[lv = rv])$. In other words, the memory is updated by assigning the evaluation of rv to the address obtained by evaluating lv.

- $st = bran(rv, rv', rv'')$: If $[rv] = \bot$ or $[rv'] = \bot$ or $[rv''] = \bot$, $s \to$ **STOP**. Otherwise, if $[rv] = 0$, $s \to ([rv''], \sigma, \mu)$, else $s \to ([rv'], \sigma, \mu)$.

- $st = malloc(lv, rv, rv')$: If $\langle lv \rangle = \bot$ or $\langle lv \rangle \notin Dom(\mu)$ or $[rv] = \bot$ or $[rv'] = \bot$, $s \to$ **STOP**. Otherwise, let μ' be any memory such that $\exists \mu'' \in Mem \cdot \exists a \in \mathbb{Z} \cdot (\mu, [rv]) \xrightarrow{new} (\mu'', a) \wedge mem' = \mu''[\langle lv \rangle = a]$. Then, $s \to ([rv'], \sigma, \mu')$.

- $st = free(rv, rv')$: If $[rv] = \bot$ or $[rv] \notin Dom(\mu)$ or $[rv'] = \bot$, $s \to$ **STOP**. Otherwise, $s \to ([rv'], \sigma, del(\mu, [rv]))$.

- $st = call(p, rv)$: Recall that p is a procedure. If $p \notin prs$ or $[rv] = \bot$, $s \to$ **STOP**. Otherwise, let $p = (GVar, LVar, Comm, ILoc)$, where $LVar = \{v_1, \ldots, v_n\}$. Let $\mu_1, \ldots, \mu_n, \mu'$ be a sequence of memories and $a_1, \ldots, a_n, a'$ be a sequence of addresses such that

  1. $(\mu, 1) \xrightarrow{new} (\mu_1, a_1)$
  2. $\forall i \in \{2, ..., n\} \cdot (\mu_{i-1}, 1) \xrightarrow{new} (\mu_i, a_i)$
  3. $(\mu_n, 1) \xrightarrow{new} (\mu', a')$
  4. $val(\mu')(a') = [rv]$

  Also, let $\sigma_1, \ldots, \sigma_n, \sigma'$ be the sequence of stores such that

  1. $\sigma_1 = push(\sigma, v_1, a_1)$
  2. $\forall i \in \{2, ..., n\} \cdot \sigma_i = push(\sigma_{i-1}, v_i, a_i)$
  3. $\sigma' = push(\sigma_n, retpc, a')$

  Then, $s \to (ILoc, \sigma', \mu')$.

- $st = ret$: If $[retpc] = \bot$, $s \to$ **STOP**. Otherwise, let a be the top element of $\sigma(retpc)$. Then, $s \to ([retpc], pop(\sigma, retpc), del(\mu, a))$.

3.3 Interpretation

In essence, the interpretation step adds extra code to the target C program P to model the manipulation of the memory by various statements of P. We have implemented the interpretation step by extending the CIL tool [Necula 02]. In addition to supporting many of the esoteric features of C, CIL provides us access to the complete type information associated with P. Also, we assume that P is preprocessed. Recall that we have modeled memory as consisting of five components: *val*, *id*, *base*, *alloc*, and *size*. The program itself manipulates *val* directly, since program assignments modify values of variables. Thus, we have to add extra code to model the manipulation of the remaining memory components. Let $P = (prs, ipr)$. First, we add four global arrays `id`, `base`, `alloc` and `size` that will be used to model *id*, *base*, *alloc*, and *size*, respectively. We also add a global variable `ID` that stores the ID of the next memory block allocated. Figure 1 shows a program that we use as a running example.

```
void foo() {
    char x[20],*y;
    y = (char*)malloc(15);
    strcpy(x,’’hello ‘‘);
    strcpy(y,’’ world ...‘‘);
    strcat(x,y + 1);
}
```

Figure 1: Example Program

Global Allocation. Let $V = \cup_{p \in prs} GVar(p)$ be the set of global variables of P. We create a dummy initial procedure `dummy_main`. In the body of `dummy_main`, we first initialize `ID` to zero. We then add the following code for each $v \in V$:

```
id[&v] = ID; base[id[&v]] = &v;
alloc[id[&v]] = 1; v = 0; ID++;
```

This code allocates fresh memory for each global variable and initializes all the global variables to zero. Finally, we call the actual initial procedure *ipr*. We create `dummy_main` because *ipr* may be called recursively multiple times, and we only want to allocate memory for a global variable once.

Local Allocation. For any procedure $p \in prs$, recall that $LVar(p)$ denotes the set of local variables of p. At the beginning of p, we add the following code for each $v \in LVar(p)$:

```
id[&v] = ID; base[id[&v]] = &v;
alloc[id[&v]] = 1; ID++;
```

In essence, the above code models the allocation of fresh memory for v. Note that local variables are uninitialized.

Allocating Arrays. For statically defined arrays, we make two allocations—one for the variable that holds the initial address of the array and another for the array itself. Specifically, for each array variable v of size k, we add the following code (in `dummy_main` if v is global and at the beginning of procedure p if v is local to p):

```
id[&v] = ID; base[id[&v]] = &v;
alloc[id[&v]] = 1; ID++;
id[v] = ID; base[id[v]] = v;
alloc[id[v]] = k; ID++;
```

For global arrays, we also set `size[id[v]]` to zero, since all global arrays are filled with zeros by default. For arrays initialized explicitly, we set `alloc[id[v]]` and `size[id[v]]` based on the initializer. For example, if we have the initializer `char v[] = ‘‘hello’’`, we set both alloc[id[v]] and size[id[v]] to 6. CIL makes all the information required to do this assignment available.

For simplicity, we assume that the value of any variable can be stored in one byte of memory. In practice, we will use type information that CIL provides to allocate memory appropriately (i.e., set the value of alloc[...]). For example, suppose we have an array of int of size 10 and a variable of type int or int* requires four bytes. Then, we set alloc[...] to 4 for the array variable and to 40 for the array itself. Here's how each statement *st* of a procedure *p* is handled by the interpretation based on the type of *st*:

- $st = asgn(lv, rv, rv')$: This statement is left as is, since the only component of memory affected by this statement is *val* and the effect is modeled by the statement itself.

- $st = bran(rv, rv', rv'')$: This statement is also left as is, since it does not affect the memory at all.

- $st = malloc(lv, rv, rv')$: This statement means that we must allocate a block of *rv* bytes of memory and store the address of the first byte of the allocated block in *lv*. Hence, we replace *st* with the following code:

  ```
  lv = *; id[lv] = ID;
  base[id[lv]] = lv;
  alloc[id[lv]] = rv; ID++;
  ```

  The first assignment sets *lv* to a nondeterministic value, since the starting address of the block of memory allocated is undefined. Such assignments are handled appropriately in the subsequent verification stage. Also, since the actual contents of the allocated block of memory is undefined, so is the value of alloc[id[lv]].

- $st = free(rv, rv')$: This statement means that we must deallocate the memory represented by *rv*. Therefore, we replace *st* with the following assignment: alloc[id[rv]] = 0.

- $st = call(p, rv)$: If the called procedure *p* is defined within the target program *P*, we leave *st* unchanged. Otherwise, *st* is replaced by a set of statements that model the effect of procedure *p* on the memory. Later, we present some specific examples of *p*, along with the set of statements that are used to replace *p*.

- $st = ret$: Since we are returning from the current procedure, we must deallocate the memory for local variables. Therefore, for each $v \in LVar(p)$, we add the following assignment just before the statement *st*: alloc[id[&v]] = 0.

Interpreting Procedures. As mentioned before, a procedure call without definition is replaced with code that models the effect of this procedure on various memory components. We now give some examples of this process. We do not consider calls to malloc and free, since they are handled directly by *malloc* and *free* statements, respectively.

Interpreting strcpy. A call to `strcpy(p,q)` is replaced with the assignment `size[id[p]] = size[id[q]] - (q - base[id[q]]) + (p - base[id[p]])`.

Interpreting strncpy. A call to `strncpy(p,q,r)` is replaced with

```
if((size[id[q]] - (q-base[id[q]])) > r)
    size[id[p]] = *; else
size[id[p]] = ((size[id[q]]) -
    (q-base[id[q]])) + (p-base[id[p]]);
```

To understand the above two interpretations, consider the following excerpt from the Linux manual pages:[5]

> The strcpy(dest,src) function copies the string pointed to by src (including the terminating null character) to the array pointed to by dest. The strncpy(dest,src,n) function is similar, except that not more than n bytes of src are copied. Thus, if there is no null byte among the first n bytes of src, the result will not be null-terminated. In the case where the length of src is less than that of n, the remainder of dest will be padded with null bytes.

We leave it to the reader to verify that the interpretations described for `strcpy` and `strcat` correspond to their specifications as shown above.

Interpreting strcat. A call to `strcat(p,q)` is replaced with the assignment `size[id[p]] = size[id[q]] - (q - base[id[q]]) + (size[id[p]] - 1)`.

Interpreting strncat. A call to `strncat(p,q,r)` is replaced with

```
if ((size[id[q]]-(q-base[id[q]]+1))>r)
  size[id[p]] = size[id[p]] + r;
else size[id[p]] = size[id[p]] +
  (size[id[q]] - (q-base[id[q]]+1));
```

In the above code, we assume that `q` is a pointer variable. If `q` is a string constant, we eliminate `(q - base[id[q]])` and replace `size[id[q]]` with the number of characters in `q` including the terminating null character. For instance, a call to `strcpy(p,''hello'')` is replaced with the assignment `size[id[p]] = 6 - (p - base[id[p]])`. Calls to other procedures that manipulate strings are treated in an appropriate manner. It is noteworthy that our interpretation scheme handles most routines in the Microsoft Strsafe[6] library in addition to the standard C string library.

Interpreting the Target Operation. Recall that the input to interpretation is a program P and a target operation O and that O leads to the addition of an assertion A. In essence, A

[5] http://jamesthornton.com/linux/man/strcpy.3.html.

[6] http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsecure/html/strsafe.asp.

asserts that if O affects the size of any buffer b, the new size of b does not exceed the number of bytes allocated for b. For instance, in each procedure call described above, if the call is also the target operation, we specify the assertion `assert(size[id[p]] <= alloc[id[p]])` in addition to the replacement code.

Logical Memory Model. We assume a logical model of memory, which means that the result of any arithmetic involving a pointer p is assumed to point somewhere within the same block of memory. This decision has the important consequence of making our procedure conservative. Thus, we may report an overflow when it cannot happen in practice.

For instance, suppose p points to the beginning of a block b of 10 bytes of memory. Suppose our program stores 20 bytes starting from the memory address $p + 100$. Since we assume that $p + 100$ still belongs to b and b has only 10 allocated bytes, we report a buffer overflow. However, during actual execution, the memory layout may cause $p + 100$ to always point to the beginning of another block b' of 50 bytes. Thus, there would be no buffer overflows in practice. Nevertheless, such situations are unpredictable and should be identified as flaws, which is what our procedure does.

Normalization. The second consequence of using a logical memory model is a restriction on the syntax of expressions. Specifically, we do not allow any "top-level" C operators to appear as part of the index of the `id` array. Expressions that obey this restriction are said to be normalized. In practice, we always normalize expressions by recursively replacing any top-level expression of the form $e\ op\ e'$ in the index of `id` with e. For instance, the expression `id[*(x + y) + z]` is normalized to `id[*(x + y)]`. The rationale behind normalization is the logical memory model. Since pointer arithmetic does not alter the block of memory involved, in a semantic sense, the block-ID of the operand is the same as the block-ID of the result. Thus, these two block-IDs can be unified, which is precisely what normalization achieves. In addition, normalization results in an expression in normal (i.e., simplest and unique) form.

For example, the result of interpreting the program P in Figure 1 (on page 11) is shown as program $\widehat{P}$ in in Figure 2. The statements of P (specifically, the procedure calls) that are replaced during interpretation are commented out in $\widehat{P}$. We assume that the target operation is the call to `strcat`. Note that due to normalization, the expression `id[y+1]` is replaced with `id[y]` while interpreting the target. We also annotate the statements of $\widehat{P}$ with invariants generated upon successful verification. The invariants are commented and enclosed in curly braces. Multiple consecutive invariants are implicitly conjuncted. Further details about the verification procedure and invariants are presented in Sections 4 and 5, respectively.

```
void foo() {
    char x[20],*y;
    id[&x] = ID; base[id[&x]] = &x;
    alloc[id[&x]] = 1; ID++;
    id[x] = ID; base[id[x]] = x;
    alloc[id[x]] = 20; ID++;
    id[&y] = ID; base[id[&y]] = &y;
    alloc[id[&y]] = 1; ID++;
    /*{(x - base[id[x]]) == 0}*/
    /*{alloc[id[x]] == 20}*/
    //y = (char*)malloc(15);
    y = *; id[y] = ID; base[id[y]] = y;
    alloc[id[y]] = 15; ID++;
    /*{(x - base[id[x]]) +
        2*(y - base[id[y]]) == 0}*/
    /*{alloc[id[x]] == 20}*/
    //strcpy(x,’’hello ‘‘);
    size[id[x]] = 7 - (x - base[id[x]]);
    /*{size[id[x]] -
       2*(y - base[id[y]]) == 7}*/
    /*{alloc[id[x]] == 20}*/
    //strcpy(y,’’ world ...‘‘);
    size[id[y]] = 11 - (y - base[id[y]]);
    //strcat(x,y + 1);
    /*{size[id[y]] - (y-base[id[y]]) +
       size[id[x]] == 18}*/
    /*{alloc[id[x]] == 20}*/
    size[id[x]] = size[id[y]] -
        (y+1-base[id[y]]) + (size[id[x]]-1);
    /*{size[id[x]] == 16}*/
    /*{alloc[id[x]] == 20}*/
    assert(size[id[x]] <= alloc[id[x]]);
}
```

Figure 2: Result of Interpretation

4 Verification

The result of the interpretation of P and O is a new program $\widehat{P}$ containing an assertion A. We now verify that A can never be violated when $\widehat{P}$ is executed. A process of “certifying” iterative refinement, shown in Figure 3, does this verification and consists of these steps:

1. **abstraction**. We construct a finite state conservative model M of the target C program $\widehat{P}$ using a technique called predicate abstraction. Also, we construct a temporal logic specification ϕ such that if M satisfies ϕ (denoted by $M \models \phi$), the assertion A can never be violated when executing $\widehat{P}$. We proceed to Step 2.
2. **certifying verification**. We model check $M \models \phi$ using a certifying model checker. If $M \models \phi$, from the above step we know that A can never fail when $\widehat{P}$ is executed. In this case, the model checker also generates an invariant Inv, and the verification process terminates with success and returns Inv. The invariant Inv is used subsequently to generate the certificate, Cert, and the annotations for P that can be used to validate Cert. On the other hand, suppose $M \not\models \phi$. Let $\widehat{CE}$ be the counterexample returned by the model checker. We proceed to Step 3.
3. **validation**. We check if $\widehat{CE}$ is also a counterexample with respect to the original C program $\widehat{P}$. If so, the verification terminates with a failure and returns the counterexample $\widehat{CE}$. Otherwise, $\widehat{CE}$ is said to be spurious, since it is a behavior that does not belong to $\widehat{P}$ but was only introduced in the model M by the abstraction process. We proceed to Step 4.
4. **refinement**. We construct a more precise model M using the spurious $\widehat{CE}$. The new model is guaranteed not to contain $\widehat{CE}$ as an admissible behavior. We now repeat Step 2 above.

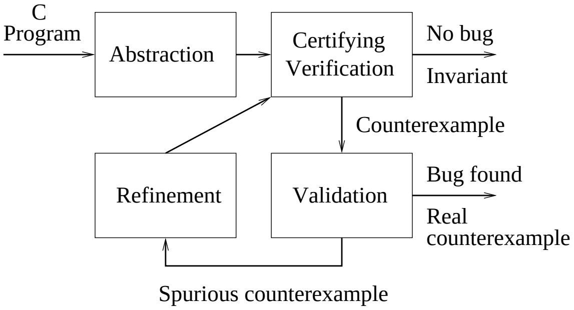

Figure 3: Certifying Iterative Refinement

Note that iterative refinement improves upon static analysis by enabling automated verification of counterexamples for spuriousness and automated model refinement to eliminate spurious counterexamples. Iterative refinement is therefore extremely suitable for detecting violations of safety conditions (such as buffer overflows) in large-scale software systems in an automated and scalable manner. In addition, the certifying iterative refinement we use can generate invariants that are used to independently validate its result. Iterative refinement, in both noncertifying [Ball 01] and certifying [Henzinger 02, Chaki 06] forms, has been described in detail in the context of other verification projects. However, there is one crucial difference between the iterative refinement (specifically, the predicate abstraction step) used in those projects and in this work. We describe the difference in more detail below:

Predicate Abstraction. Predicate abstraction [Graf 97] creates a conservative finite state machine model of a program. The basic idea behind predicate abstraction is to keep the control-flow aspect of the program's execution unchanged and to abstract the infinite possible memory configurations that the program can reach during its execution into a finite set of values for a collection of predicates. While the full details of predicate abstraction are beyond the scope of this report, an important concept involved is that of a *weakest precondition.* Indeed, this concept is also used in the validation step of iterative refinement.

Weakest Precondition. Given a memory m and an rvalue e, we say that m satisfies e, and denote this as $m \models e$, iff e evaluates to a nonzero value according to m. For instance, if e is $v > 0$, $m \models e$ iff v is assigned a positive value according to m. An rvalue e is said to be weaker than an rvalue e' iff $m \models e' \Rightarrow e$ [i.e., $m \models (!e')||e$ as per our syntax] for any m. Thus, $v < 10$ is weaker than $v < 5$. Then, the weakest precondition of an rvalue e with respect to an assignment $lhs = rhs$, denoted by $\mathcal{WP}[e]\,\{lhs = rhs\}$, is defined to be the weakest rvalue e' such that the following holds: on executing the assignment $lhs = rhs$ from any memory m' such that $m' \models e'$, a program is guaranteed to reach a memory m such that $m \models e$.

In practice, e' is usually obtained by replacing every occurrence of lhs with rhs in e. To understand why this works, let e be $v < 10$ and the assignment be $v = v' + 5$. Now, for $v < 10$ to hold after the assignment, it is necessary and sufficient for $v' + 5 < 10$ to hold before the assignment, and that is precisely the rvalue obtained by replacing v with $v' + 5$ in e. However, for our purposes, e' is computed by normalizing the result of replacing lhs with rhs in e. For instance, if e is $\texttt{base}[\texttt{id}[p]] = q$ and the assignment is $p = p + 1$, $\mathcal{WP}[e]\,\{p = p + 1\}$ is $\texttt{base}[\texttt{id}[p]] = q$. Note that due to the logical memory model, our result is equivalent to the expression (specifically, $\texttt{base}[\texttt{id}[p + 1]] = q$) that we would obtain by the standard procedure.

5 Certification

Recall in Step 2 of the iterative refinement process that if verification succeeds—the inviolability of assertion A could be proved—we obtain an invariant Inv. The process by which Inv is constructed has been described in the literature [Chaki 06] and we do not present it here. Instead, we describe how Inv is used to generate (1) a certificate Cert that proves why the operation O cannot lead to a buffer overflow and (2) annotations that enable the independent validation of Cert.

Invariant. Let $\widehat{P} = (\mathit{prs}, \mathit{ipr})$ and $L = \cup_{p \in \mathit{prs}} \mathit{Dom}(\mathit{Comm}(p))$ be the set of locations of the procedures of $\widehat{P}$. Then, $\mathit{Inv} : L \to \mathit{RV}$ is a function that maps each location l to an rvalue rv. For any location $l \in L$, let st be the statement at location l. Then, in essence, $\mathit{Inv}(l)$ expresses a condition that must hold whenever the execution of $\widehat{P}$ reaches st. For instance, suppose that $\mathit{Inv}(l) = (x < y)$. Then, the value of variable x must be less than the value of y whenever $\widehat{P}$ is about to execute the statement at l. Figure 2 shows our example $\widehat{P}$ annotated with invariants.

Certificate Construction. In essence, Cert is a proof of a verification condition VC, which is a logical statement of the following three facts:

- **(Inv1)** Inv is satisfied by the initial state of $\widehat{P}$.
- **(Inv2)** Inv is preserved by the execution of $\widehat{P}$.
- **(Inv3)** Let A assert the rvalue e and let l be the location of A.

Then, $\mathit{Inv}(l)$ implies e. Note that the facts **Inv1** and **Inv2** above assert that Inv is indeed a valid invariant for $\widehat{P}$, while **Inv3** asserts that Inv implies the inviolability of A.

To construct Cert, we use the following two steps:

- **(Cert1)** Construct VC.
- **(Cert2)** Prove VC using a proof-generating theorem prover.

For Step **Cert2**, we use a theorem prover based on Boolean satisfiability (SAT). The idea is to translate the logical negation of VC into a propositional formula Ω and check for the satisfiability of Ω using a proof-generating SAT solver like ZCHAFF [Zhang 03]. If Ω is unsatisfiable (which is equivalent to VC being valid), Cert is the resolution proof emitted by the SAT solver. Such a SAT-based approach yields extremely compact certificates in practice and is described in detail elsewhere [Chaki 06]. We now describe Step **Cert1**—that is, the procedure for constructing VC—beginning with a generalization of the concept of weakest preconditions to statements.

Statement Precondition. For any invariant Inv and any statement st, we write $\mathit{Pre}(\mathit{st}, \mathit{Inv})$ to mean the weakest condition that must hold before the execution of st in order for the invariant to be satisfied at each of the successor locations of st. For instance, suppose that st is $\mathit{asgn}(\mathit{lv}, \mathit{rv}, \mathit{rv}')$. Then, $\mathit{Pre}(\mathit{st}, \mathit{Inv})$ is simply $\mathcal{WP}[\mathit{Inv}(\mathit{rv}')]\,\{\mathit{lv} = \mathit{rv}\}$. Also, if st is

$bran(rv, rv', rv'')$, $Pre(st, Inv) = (rv \;\&\&\; Inv(rv')) \;||\; (!rv \;\&\&\; Inv(rv''))$. For other types of statements, $Pre(st, Inv)$ is defined appropriately. Henceforth, we use the terms *formula* and *rvalue* synonymously.

Constructing VC. We generate VC by conjunction over a set Ψ of formulas defined as follows:

- Let $l = ILoc(ipr(\widehat{P}))$ be the initial location of the initial procedure of $\widehat{P}$. Note that $Inv(l)$ is logically equivalent to fact **Inv1** above. Then, $Inv(l) \in \Psi$.
- For each statement st of $\widehat{P}$, let l be the location of st. The formula $\phi = (Inv(l) \Rightarrow Pre(st, Inv))$ asserts that Inv is preserved by the execution of st. Then, $\phi \in \Psi$.
- Let A assert the rvalue e and let l be the location of A. Note that the formula $\phi = (Inv(l) \Rightarrow e)$ is equivalent to **Inv3** above. Then, $\phi \in \Psi$.

Example. With respect to Figure 2, Ψ contains the following formula, which states the inviolability of the assertion and is clearly valid:

$$(\texttt{size}[\texttt{id}[\texttt{x}]] == 16)\&\&(\texttt{alloc}[\texttt{id}[\texttt{x}]] == 20)$$
$$\Rightarrow (\texttt{size}[\texttt{id}[\texttt{x}]] <= \texttt{alloc}[\texttt{id}[\texttt{x}]])$$

We leave it up to you to verify that the other elements of Ψ are also valid formulas.

Annotations. Recall that, in addition to $Cert$, we generate annotations for the original program P that can be used to validate $Cert$. These annotations are simply rvalues associated with the statements of P and are derived from the invariant Inv in the following manner. Consider any statement st of P. Suppose that st is also a statement of the interpreted program $\widehat{P}$. Let l be the location of st. Then, we annotate st with the rvalue $Inv(l)$. Otherwise, we know that st was replaced in $\widehat{P}$ with a sequence of statements X. Let $l_1, \ldots, l_n$ be the locations of the statements in X. Then, we annotate st with the sequence of lvalues $Inv(l_1), \ldots, Inv(l_k)$.

Certificate Validation. The annotations generated above are used to validate $Cert$ in the following manner:

1. **(Step 1)** First, we use interpretation to reconstruct $\widehat{P}$ from P. Note that doing so is possible, since interpretation is a completely deterministic procedure and is guaranteed to yield the same output given identical inputs.
2. **(Step 2)** Next, using the annotations, we reconstruct the invariant Inv. Once again, this is possible since the annotations are generated in a deterministic manner. Also, for the same P and $\widehat{P}$ there is a one-to-one mapping between invariants and annotations.
3. **(Step 3)** Once we have Inv, we reconstruct VC using $\widehat{P}$ and Inv. Due to the deterministic nature of the procedure for computing VC, we are assured of obtaining the exact same VC obtained during the generation of $Cert$.

4. **(Step 4)** Finally, we use a suitable proof checker to confirm that *Cert* is a proper proof of the validity of VC.

Trusted Computing Base. The trusted computing base (TCB) required for validating *Cert* consists of the tools needed to perform Steps 1 to 4 described above. Specifically, it contains the

- interpreter needed for Step 1
- program for reconstructing *Inv* from the annotations
- VC generator
- proof checker required in Step 4

Thus, the TCB is much smaller and simpler (and hence more trustworthy or amenable to formal verification) than the software model checker required to verify $\widehat{P}$ or the theorem prover necessary for proving VC. Indeed, measurements on our own infrastructure showed that the TCB is about 15 times smaller than the rest of the system.

6 Experimental Validation

We implemented our approach on top of the ComFoRT reasoning framework and the CIL tool. Specifically, we implemented interpretation as an extension of CIL, while the verification and certification stages were implemented by extending ComFoRT. We experimented with a set of four publicly available benchmarks. All our experiments were carried out on a quad 2.4 GHz machine with 4GB RAM running RedHat Linux 9.

Tool	FlawFinder		ITS4		RATS		Splint		BOON		ComFoRT	
	#	%	#	%	#	%	#	%	#	%	#	%
Actual Errors	10	91	11	100	11	100	3	27	4	36	11	100
False Alarms	6	67	4	44	6	67	0	0	3	33	0	0
Safe Proved	3	33	5	56	3	33	9	100	6	67	9	100
Missed Errors	1	9	0	0	0	0	8	73	7	64	0	0

Figure 4: Wilander Suite Results

Wilander Suite. The first set of benchmarks was designed by Wilander and Kamkar to compare a set of publicly available static intrusion-detection tools (FlawFinder, ITS4, RATS, Splint, and BOON) [Wilander 02]. The suite consisted of a set of procedures to check for buffer-overflow and format-string vulnerabilities. For our experiments, we used only the subset of benchmarks targeted toward overflows containing 9 safe and 11 buggy (i.e., unsafe) examples. Our results are summarized in Figure 4 under the ComFoRT column. The remaining results are reproduced from those of Wilander and Kamkar [Wilander 02]. The subcolumns under "#" contain actual numbers, while those under "%" contain percentages. We see that our approach (ComFoRT) is the only one with zero false positives (i.e., false alarms) and zero false negatives (i.e., missed errors). Moreover, the size of *Cert* (625 bytes) was negligible compared to the benchmark file size (14KB).

Tool	TD	DR	FA	FAR	TC	CR
ARCHER	264	90.7	0	0	0	0
BOON	2	0.69	0	0	0	0
PolySpace	290	99.7	7	2.4	7	2.4
Splint	164	56.4	35	12	35	12
UNO	151	51.9	0	0	0	0
ComFoRT	208	71.5	0	0	0	0

Figure 5: Kratciewicz Suite Results

Kratciewicz Suite. The second set of benchmarks was developed by Kratciewicz [Kratciewicz 05]. This set is similar to Wilander's but is more comprehensive. Specifically, it consists of 291 test cases involving illegal array accesses, shared memory, signal handling, and function pointers. These benchmarks were also used originally to compare a suite of public tools, namely, ARCHER, BOON, PolySpace, Splint, and UNO. The results of our approach on these benchmarks are summarized in Figure 5 in the row labelled ComFoRT. The other rows are reproduced from Kratciewicz's thesis. The columns in Figure 5 have the following meaning: TD = number of examples handled; DR = rate of handling examples; FA =

number of false alarms; FAR = false alarm rate; TC = number of confusions; CR = confusion rate. Our approach yields zero false alarms and zero confusions (i.e., the inability to distinguish between correct and buggy versions of the same example) and is second only to ARCHER overall. Also, the average size of *Cert* (56 bytes) was negligible compared to the benchmark file size (3KB). The main reason for our partial coverage of the benchmarks is the inability of our model checker to handle features of C, such as function pointers.

Category	**TP**	**TPD**	**TN**	**TNC**
Strings	14	10	11	7
Pointer Subterfuge	1	1	1	1
Dynamic Memory	4	4	2	2

Figure 6: Seacord Suite Results

Seacord Suite. We derived the third set of our benchmarks from examples in a book on secure programming in C/C++ [Seacord 06]. We selected a set of 31 code snippets dealing with string buffer overflow, pointer vulnerabilities, and dynamic memory allocation. The results of our experiments with this benchmark suite are presented in Figure 6. The columns in the figure should be read as follows: TP = true positives; TPD = true positives detected; TN = true negatives; TNC = true negatives confirmed. Our approach was perfect on examples involving pointer subterfuge and dynamic memory allocation. However, it could not handle some of the string examples due to limitations of the model checker. The average file size for the correct (i.e., safe) examples was over 1.3KB, while the average size of *Cert* was just 71 bytes.

Tool	**P(d)**	**P(f)**	$\mathbf{P(\neg f \mid d)}$
PolySpace	0.87	0.5	0.37
Splint	0.57	0.43	0.30
BOON	0.05	0.05	-
ARCHER	0.01	0	-
UNO	0	0	-
ComFoRT	0.43	0.21	1.00

Figure 7: Zitser Suite Results

Zitser Suite. The final set of our benchmarks was developed by Zitser and colleagues to test the same set of tools that Kratciewicz used for her thesis [Zitser 04]. This test suite was the most realistic and consisted of code from real-life software (bind, sendmail and wu-ftpd) with known vulnerabilities. Zitser, Lippmann and Leek created 14 buggy examples (4 from bind, 7 from sendmail, and 3 from wu-ftpd) that contained actual vulnerabilities previously found in these programs. For each buggy example, they also created a correct version by applying the patches used in reality to fix these errors.

The results of our technique are summarized in Figure 7 in the ComFoRT row. As usual, the other rows are reproduced from those of Zitser, Lippmann, and Leek [Zitser 04]. The columns in Figure 7 can be understood as follows: Let $T(d)$ be the maximum number of overflow detections possible and $C(d)$ be the number of overflows actually detected. Let $T(f)$ and $C(f)$ be the corresponding numbers for false alarms. Then, $P(d) = C(d)/T(d)$ and $P(f) = C(f)/T(f)$. Also, let $C(df)$ be the number of times a detection was paired with a false alarm (i.e., a confusion) for a given buggy/correct pair of programs. Then, $P(\neg f|d) = 1 - C(df)/C(d)$. We note that our approach has the lowest false alarm rate (we ignore the results for BOON, ARCHER, and UNO, since they were insignificant [Zitser 04]). Moreover, we have a perfect 1.0 score for $P(\neg f|d)$, meaning that when our approach found an actual buffer overflow in a program, it could also prove the fixed version of the same program to be correct. Finally, the average size of *Cert* was just 172 bytes against the average benchmark file size of 12KB.

7 Conclusion

We presented a framework for finding and certifying the absence of buffer overflows in C programs. Our approach is based on a combination of a novel interpretation technique and certifying software model checking. Experimental results with an early prototype indicate that our technique is effective on a set of public and real-life benchmarks. Most of the issues we faced were due to limitations of the model checker—limitations that must be addressed to make our approach more widely adoptable. Finally, a more general memory model and interpretation procedure are needed to extend our approach to a wider class of vulnerabilities such as format string errors.

References

[Ball 01] Ball, T. & Rajamani, S. K. “Automatically Validating Temporal Safety Properties of Interfaces”, 103–122. *Proceedings of the 8th International SPIN Workshop on Model Checking of Software (SPIN ’01)*, volume 2057 of *Lecture Notes in Computer Science*. Toronto, Canada, May 19–20, 2001. New York, NY: Springer-Verlag, 2001.

[Chaki 05] Chaki, S.; Ivers, J.; Sharygina, N.; & Wallnau, K. “The ComFoRT Reasoning Framework”, 164–169. *Proceedings of the 17th International Conference on Computer Aided Verification (CAV ’05)*, volume 3576 of *Lecture Notes in Computer Science*. Edinburgh, Scotland, July 6–10, 2005. New York, NY: Springer-Verlag, 2005.

[Chaki 06] Chaki, S. “SAT-Based Software Certification”, 151–166. *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS ’06)*, volume 3920 of *Lecture Notes in Computer Science*. Vienna, Austria, March 25–April 2, 2006. Berlin, Germany: Springer-Verlag, 2006.

[Cowan 00] Cowan, C.; Wagle, P.; Pu, C.; Beattie, S.; & Walpole, J. “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade”, 119–129. *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*. Hilton Head, South Carolina, January 25–27, 2000. Los Alamitos, CA: IEEE Computer Society, 2000.

[Dahn 03] Dahn, C. & Mancoridis, S. “Using Program Transformation to Secure C Programs Against Buffer Overflows”, 323–333. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE ’03)*. Victoria, BC, Canada, November 13–16, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.

[Dhurjati 06] Dhurjati, D. & Adve, V. S. “Backwards-Compatible Array Bounds Checking for C with Very Low Overhead”, 162–171. *Proceedings of the 28th International Conference on Software Engineering (ICSE ’06)*. Shanghai, China, May 20–28, 2006. New York, NY: Association for Computing Machinery, 2006.

[Dor 03] Dor, N.; Rodeh, M.; & Sagiv, S. “CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C”, 155–167. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI ’03)*. San Diego, CA, June 9–11, 2003. New York, NY: Association for Computing Machinery, 2003.

[Ganapathy 03] Ganapathy, V.; Jha, S.; Chandler, D.; Melski, D.; & Vitek, D. “Buffer Overrun Detection Using Linear Programming and Static Analysis”, 345–354. *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS ’03).* Washington, DC, October 27–30, 2003. New York, NY: Association for Computing Machinery, 2003.

[Graf 97] Graf, S. & Saïdi, H. “Construction of Abstract State Graphs with PVS”, 72–83. *Proceedings of the 9th International Conference on Computer Aided Verification (CAV ’97)*, volume 1254 of *Lecture Notes in Computer Science.* Haifa, Israel, June 22–25, 1997. New York, NY: Springer-Verlag, 1997.

[Henzinger 02] Henzinger, T. A.; Jhala, R.; Majumdar, R.; Necula, G. C.; Sutre, G.; & Weimer, W. “Temporal-Safety Proofs for Systems Code”, 526–538. *Proceedings of the 14th International Conference on Computer Aided Verification (CAV ’02)*, volume 2404 of *Lecture Notes in Computer Science.* Copenhagen, Denmark, July 27–31, 2002. New York, NY: Springer-Verlag, 2002.

[Jarzombek 04] Jarzombek, J. “Systems, Networks and Information Integration Context for Software Assurance”. http://www.sei.cmu.edu/products/events/acquisition/2004-presentations/jarzombek/jarzombek.pdf, (January 2004).

[Jones 97] Jones, R. & Kelly, P. “Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs”, 13–26. *Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG ’97)*, volume 2(009) of *Linkoping Electronic Articles in Computer and Information Science.* Linkoping, Sweden, May 26–27, 1997. Linkoping, Sweden, 1997.

[Kratciewicz 05] Kratciewicz, K. “Evaluating Static Analysis Tools for Detecting Buffer Overflows in C Code”. Master’s diss., Harvard University, Cambridge, MA, 2005.

[Namjoshi 01] Namjoshi, K. S. “Certifying Model Checkers”, 2–13. *Proceedings of the 13th International Conference on Computer Aided Verification (CAV ’01)*, volume 2102 of *Lecture Notes in Computer Science.* Paris, France, July 18–22, 2001. New York, NY: Springer-Verlag, 2001.

[Namjoshi 03] Namjoshi, K. S. “Lifting Temporal Proofs through Abstractions”, 174–188. *Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI ’03)*, volume 2575 of *Lecture Notes in Computer Science.* New York, NY, January 9–11, 2002. New York, NY: Springer-Verlag, 2003.

[Necula 97] Necula, G. C. "Proof-Carrying Code", 106–119. *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97).* Paris, France, January 15–17, 1997. New York, NY: Association for Computing Machinery, 1997.

[Necula 02] Necula, G. C.; McPeak, S.; Rahul, S. P.; & Weimer, W. "CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs", 213–228. *Proceedings of the 11th International Conference on Compiler Construction (CC '02)*, volume 2304 of *Lecture Notes in Computer Science.* Grenoble, France, April 8–12, 2002. New York, NY: Springer-Verlag, 2002.

[NIST 02] National Institute of Standards and Technology. "Planning Report 02-3: The Economic Impacts of Inadequate Infrastructure for Software Testing." http://www.nist.gov/director/prog-ofc/report02-3.pdf (May 2002).

[Ruwase 04] Ruwase, O. & Lam, M. S. "A Practical Dynamic Buffer Overflow Detector", 159–169. *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04).* San Diego, CA, February 5–6, 2004. Reston, VA: Internet Society, 2004.

[Schmidt 98] Schmidt, D. A. "Data Flow Analysis is Model Checking of Abstract Interpretations", 38–48. *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98).* San Diego, CA, January 19–21, 1998. New York, NY: Association for Computing Machinery, 1998.

[Seacord 06] Seacord, R. *Secure Coding in C and C++.* Upper Saddle River, NJ: Addison-Wesley, 2006.

[Shankar 01] Shankar, U.; Talwar, K.; Foster, J. S.; & Wagner, D. "Detecting Format String Vulnerabilities with Type Qualifiers", 201–216. *Proceedings of the 10th USENIX Security Symposium.* Washington, D.C., August 13–17, 2001. Berkeley, CA, 2001.

[Wagner 00] Wagner, D.; Foster, J. S.; Brewer, E. A.; & Aiken, A. "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities". *Proceedings of the 7th Annual Network and Distributed System Security Symposium (NDSS '00).* San Diego, CA, February 2–4, 2000: Internet Society, 2000.

[Wilander 02] Wilander, J. & Kamkar, M. "A Comparison of Publicly Available Tools for Static Intrusion Prevention", 68–84. *Proceedings of the 7th Nordic Workshop on Secure IT Systems (NORDSEC '02).* Karlstad, Sweden, November 7–8, 2002. Karlstad, Sweden: Karlstad University Press, 2002.

[Zhang 03] Zhang, L. & Malik, S. "Validating SAT Solvers Using an Independent Resolution-Based Checker: Practical Implementations and Other Applications", 10880–10885. *Proceedings of 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003).* Munich, Germany, March 3–7, 2003. Los Alamitos, CA: IEEE Computer Society, 2003.

[Zitser 04] Zitser, M.; Lippmann, R.; & Leek, T. "Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code", 97–106. *Proceedings of the 12th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE '04).* Newport Beach, CA, October 31–November 5, 2004. New York, NY: Association for Computing Machinery, 2004.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY
(Leave Blank)

2. REPORT DATE
September 2006

3. REPORT TYPE AND DATES COVERED
Final

4. TITLE AND SUBTITLE
Certifying the Absence of Buffer Overflows

5. FUNDING NUMBERS
FA8721-05-C-0003

6. AUTHOR(S)
Sagar Chaki & Scott Hissam

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2006-TN-030

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. abstract (maximum 200 words)

Despite increased awareness and efforts to reduce buffer overflows, they continue to be the cause of most software vulnerabilities. In large part, these problems are due to the widespread use of unsafe library routines among programmers. For reasons like efficiency, such routines will continue to be used, even during the development of mission-critical and safety-critical software systems. Effective certification techniques are needed to ascertain whether unsafe routines are used in a safe manner.

This report presents a technique for certifying the safety of buffer manipulations in C programs. The approach is based on two key ideas: (1) using a certifying model checker to automatically verify that a buffer manipulation is safe and (2) validating the resulting invariant and proving it with a decision procedure based on Boolean satisfiability. This report also discusses the advantages and limitations of the approach with respect to today's existing solutions for buffer-overflow detection. Experimental results are presented that position the technique favorably against other static overflow-detection tools and indicate that the procedure can complement and augment these tools from a purely verification perspective.

14. SUBJECT TERMS
software validation, security, buffer overflow, model checking

15. NUMBER OF PAGES
38

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT
Unclassified

18. SECURITY CLASSIFICATION OF THIS PAGE
Unclassified

19. SECURITY CLASSIFICATION OF ABSTRACT
Unclassified

20. LIMITATION OF ABSTRACT
UL